GPU Architectures

Maurizio Cerrato
@speedwago

About me & legal bits

- Principal Graphics engineer in Sony Interactive Entertainment.
- All the informations in this talk are publicly available.
- Disclaimer:
  All views expressed on this talk are of my own and they do not represent the opinion of Sony Interactive Entertainment.

Why we are here

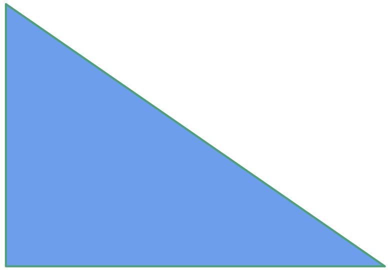
- To understand the fundamentals on how a Gpu work
- Will help you to understand performance issues
- To share ideas

Overview

- Quick review of the graphics pipeline
- Mapping the graphics pipeline into the gpu blocks
- How a shader core works
- Some real gpu use cases
- Mobile Gpus
- Conclusions

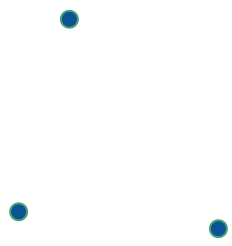
Rasterization in six slides (I)

- Before we start we need to understand the problem we want to solve
- Turning triangle data into pixels
- Many steps involved
  - Geometry processing
    - Project triangles in screen space
- Rasterization
  - Find the the pixel covered by triangle
    - Or triangle walking
- Pixel processing
  - Actually assign a color to the pixel



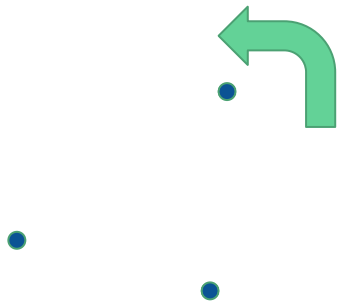
Rasterization in six slides (II)

- In the beginning we only have vertex data
- 3d point coordinates



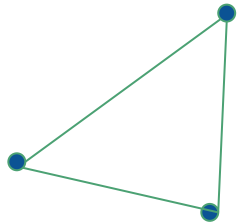
Rasterization in six slides (III)

- Vertex are transformed and projected in 2d space
- We call this “vertex shading”



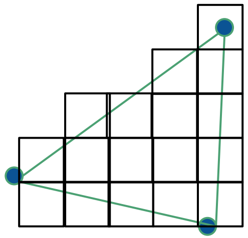
Rasterization in six slides (IV)

- They are then assembled into a primitive



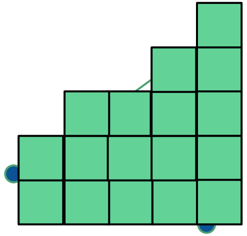
Rasterization in six slides (V)

- Then we determine what pixels of the screen the primitive “touches”
- We call this “fragments”



Rasterization in six slides (VI)

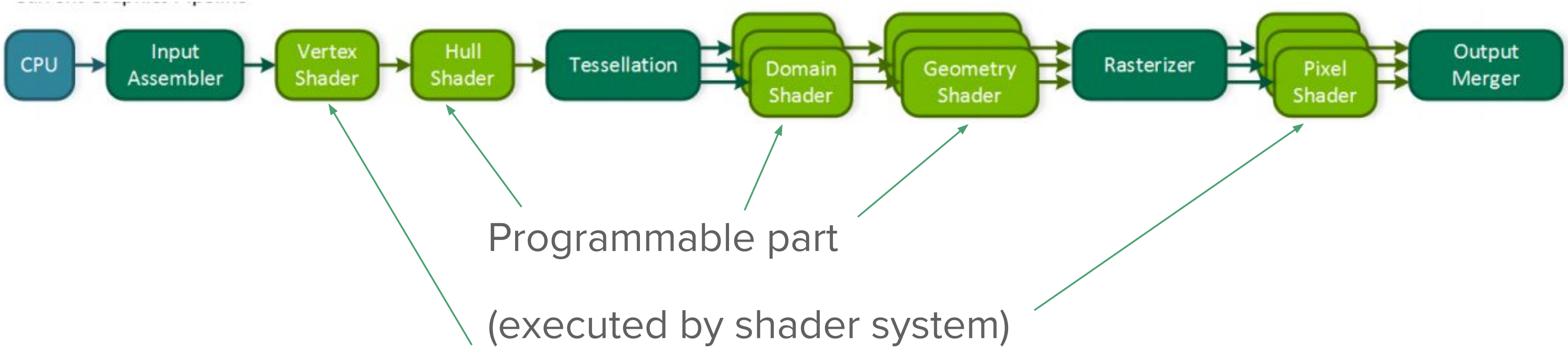
- Finally we need to assign a color to each one of them
- We call this “Fragment shading”



Gpu to help

- A modern gpu can accelerate all of this
  - Wasn’t always the case in the past, but that’s another story
- All of the previous operation map to several specific HW block
- Some functionality are programmable and performed by the shader cores
  - Ie. fragment shader, vertex shader
- Other are fixed but parameterizable
  - ie. Primitive assembly, blending.

Graphics pipeline



- "Logical" pipeline described in OGL/DX specification
  - It's an abstraction
- At physical level things are very different
  - As long as specs are met there's no problem
- Today we will look at things from a slightly closer POV

Anatomy of a GPU

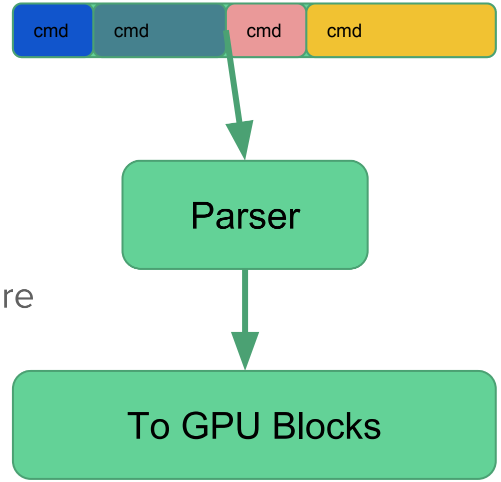
- Extremely Parallel machine
    - Thousands of “threads” in flight
    - But
        - Limited flow control
        - Some threads shares program counter
        - No Inter process communication
    - Extremely good at doing lots of independent operations at the same time
- Memory bandwidth is very high
    - Hundreds of GB/s
    - But
        - Very high latency
            - Thousand of cycles
        - Latency hidind mechanism necessary
- Graphics pipeline is organized to overcome those constraints

Before all that

- CPU issues commands to the GPU
  - eg:
    - Draw using those vertices and indices
    - Set this viewport
    - Changing states
    - May contain constants for shaders
    - Blend everything over using this blending function
- Command are not executed immediately
  - Typically the cpu prepare command for the next frame while the GPU is rendering the current one
    - Double buffering
  - The commands written into a command buffer
  - The GPU parse them

Introducing Command Buffer Parser

- Parse the command buffer
- Send commands down the graphics pipeline
- Synchronization point between cpu and gpu
    - Surface synch
        - Eg: wait that all the draw command on that rt finished before binding it as texture
- Cpu bound applications: command buffer is not filled fast enough and Gpu is idle.



Geometry stage

Lots of sub stages

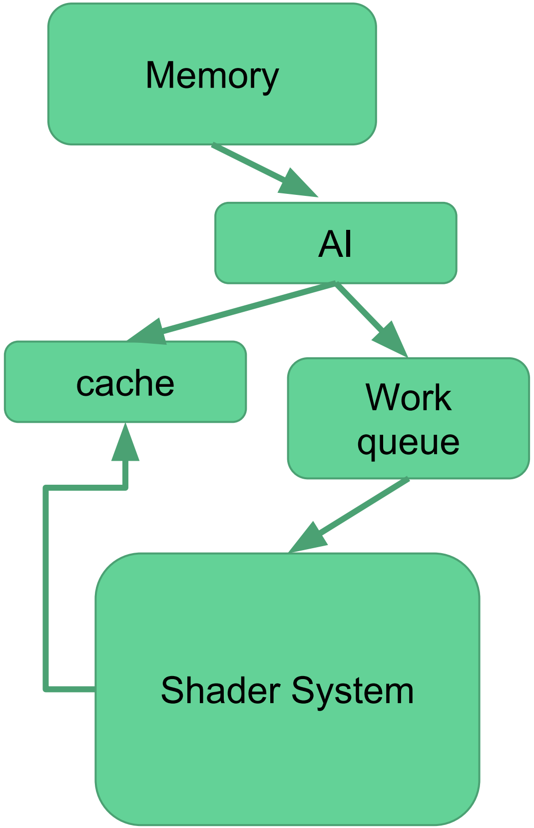
- Input assembly
- Vertex shading
- Primitive assembly

Plus optional stuff:

- Domain shader
- Tessellation shader
- Geometry shader
- Stream out
- For simplicity we are skipping those

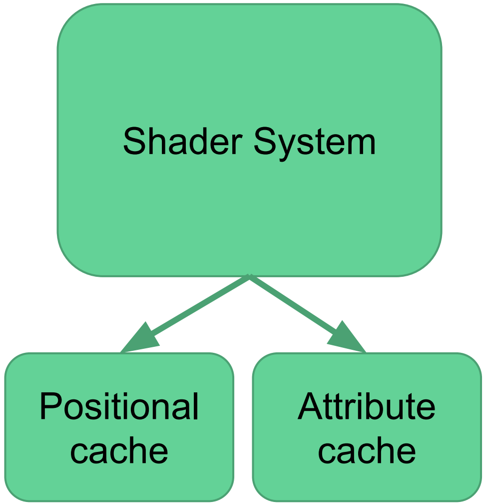
Input assembly Unit

- Fetches the indices / vertex from main memory
- Has a vertex reuse cache
  - Triangles share vertices , so it is likely to have cache hit.
  - Cache miss means the vertex need to be sent to the shader system to transform
- When enough cache misses are accumulated a job is sent to the shader core
- Usually there are more than one input assembly unit in a GPU.
  - Work distribution is usually done at drawcall level
    - Eg, assign 128 indices to a different AI unit.



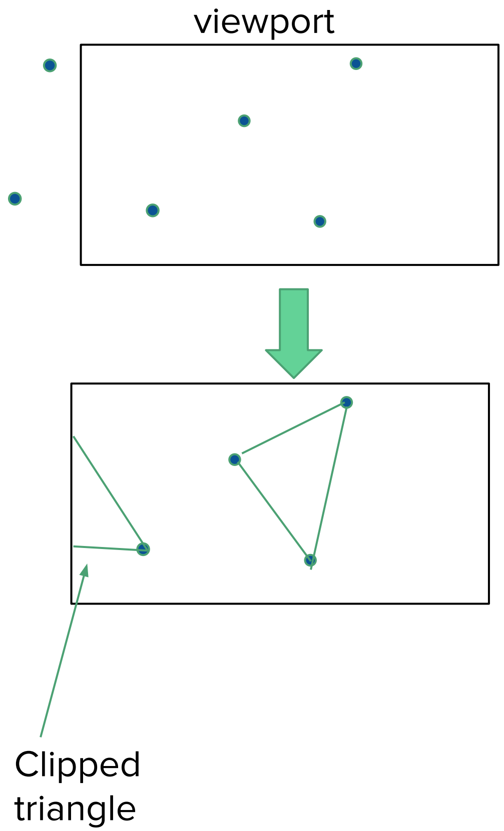
Vertex shading

- Done by shader Core
  - Details later
- First stage that is entirely programmable
- Export position and vertex attribute to forward to pixel shading
- Positions are stored in a positional cache, used in primitive assembly/setup
- Attributes are stored in a separate cache, they are needed only in pixel shaders



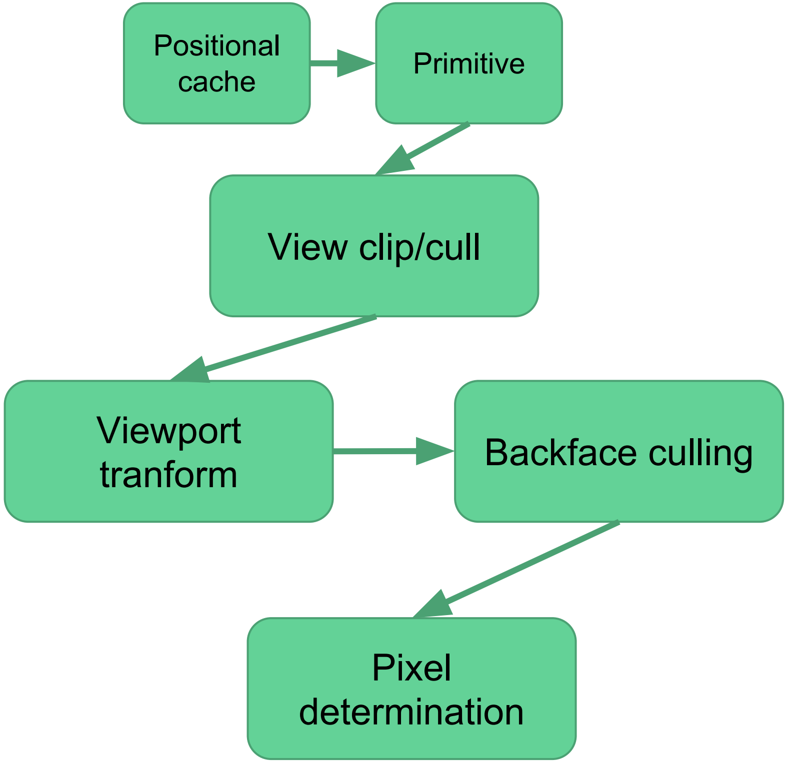
Primitive assembly Unit

- So far we have only point (vertex) transformed
- Primitive assembly takes the position form the position cache
- Use the connectivity information we gave in the API (eg Trilist)
- And turn them in to triangles
- At this point triangles need to be discarded if outside the view
- Clipped if partially in view
  - Clipping produces more triangle, expensive so guard band used to minimize



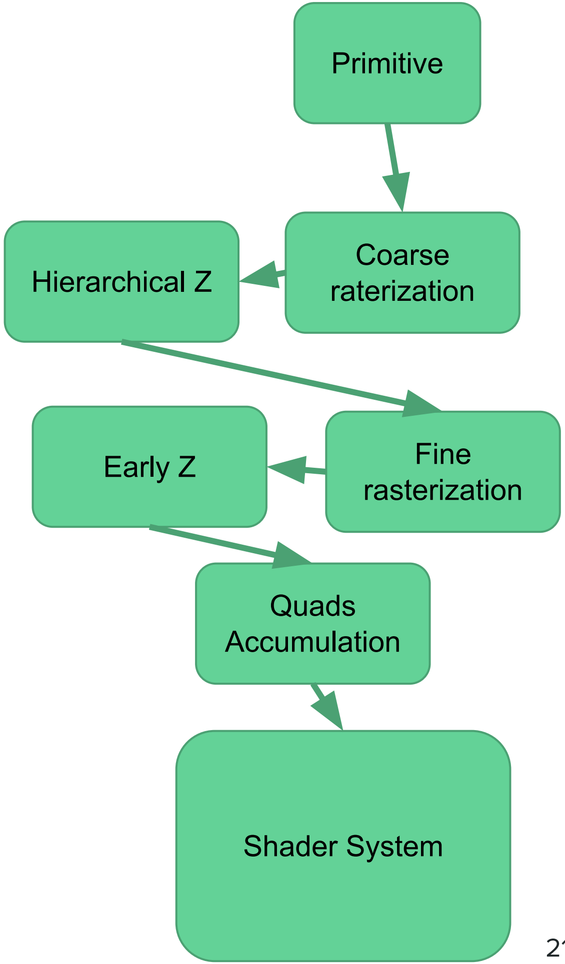
Primitive assembly Unit

- Surviving primitive are then perspective projected (divided by w) and viewport transformed
- Backfacing and zero area culling happens here
- Vertices are “snapped” into pixel
- A CPU bounding box culling can avoid PA Being overwhelmed



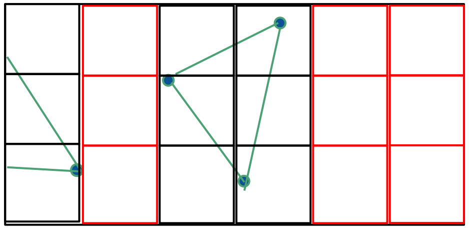
Triangle rasterization Unit

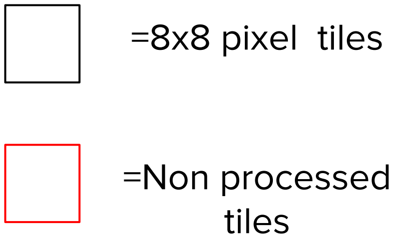
- Find which pixel covers a triangle
- Done in an hierarchical fashion, at least 2 level
- There are many rasterizers in a GPU, each one serving a portion of the screen
- perform hierarchical Z and early Z
- Assembles quad (2x2) pixels
- When enough quads are accumulated a job is sent to the shader system



Coarse rasterization

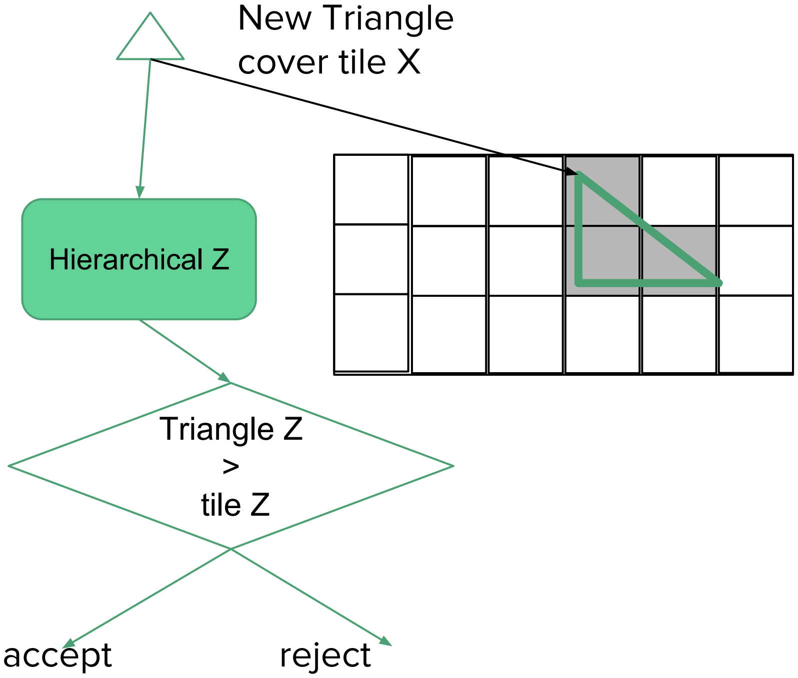
- Screen is divided in “tiles”
- Triangle is first tested against those tile
- If the triangle doesn't hit the tile then we saved unnecessary tests





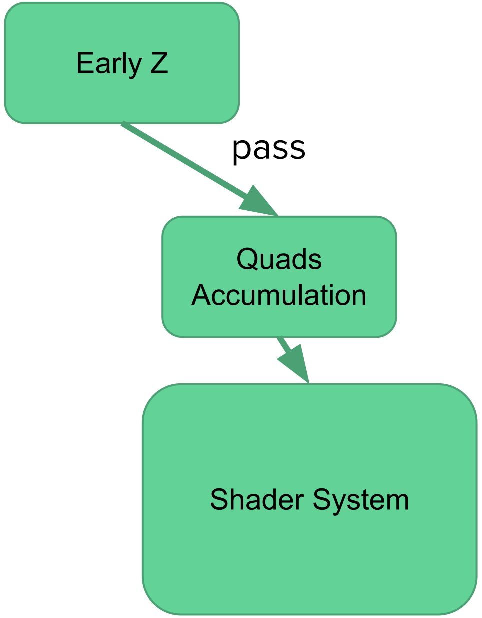
Hierarchical Z unit

- Perform early rejection of the primitive
- For each tile
  - Keep track of the current min and max z in tiles
  - If the triangle min z is larger than tile max z
  - Reject the triangle
  - Otherwise update the min max
- Also handling fast z clears
- Can skip fragment processing of entire triangles
- Remember : triangles are never sorted anywhere they are processed in submission order!



Early Z

- Similar to Hiz
- But Done at sample level
- Compute the depth of the pixel before of its color
- Pixel shader is not executed if the test don’t pass
- Not always possible
  Ie. Tralucent , discard

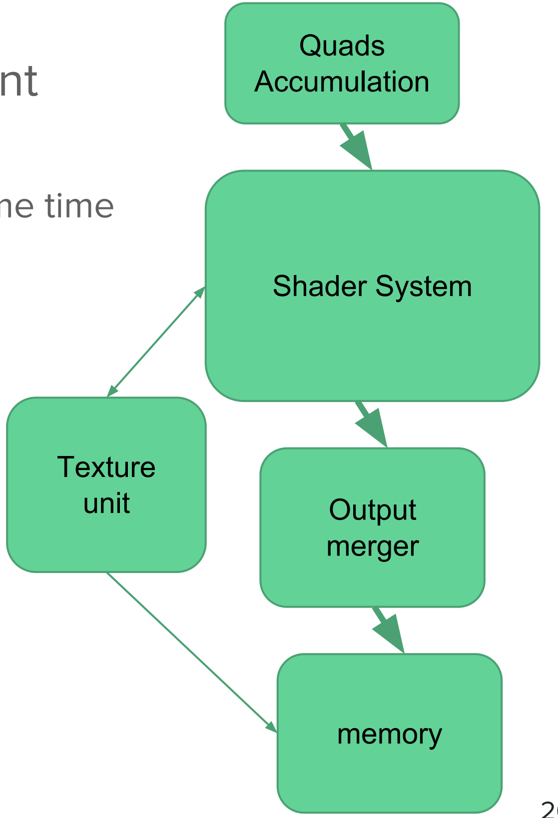


Depth compression

- If a triangle touches more pixels it’s expensive to store its depth as a float
- Instead you can use 3 float to identify the whole plane of the triangles.
- Example: tile is 8x8 pixels x 4 bytes float is 256 bytes of uncompressed depth
    - If we use plane compression best case is 12 bytes (1 triangles cover the whole tile)
    - At some point in this case after 21 planes, it start to have the same footprint.
- Greatly reduce memory bandwidth for big triangles
    - Small triangles -> more bandwidth

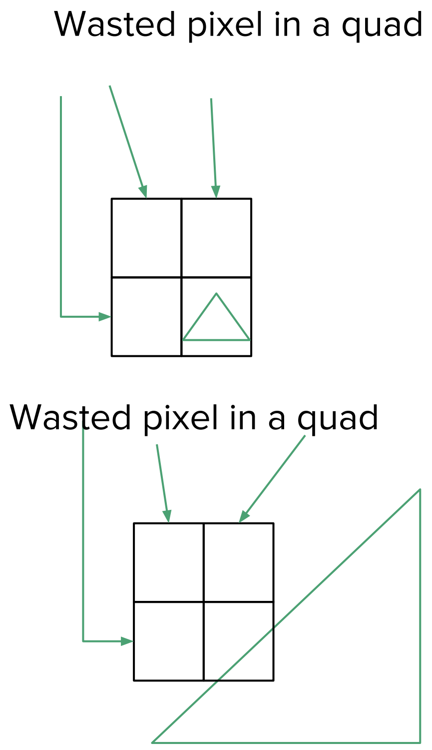
Fragment shading

- Once the rasterize packed the enough quads, fragment shader can be dispatched
  - Thousands of pixel shading operation can be in flight at the same time in a gpu
- Depends on the architecture usually is 16 or 8 quads dispatched together
- This task if performed by the shader system
- Quads are needed for calculation of the derivative
  - Derivative are needed for selection the mip level
  - Mips are needed for better visual quality and performance



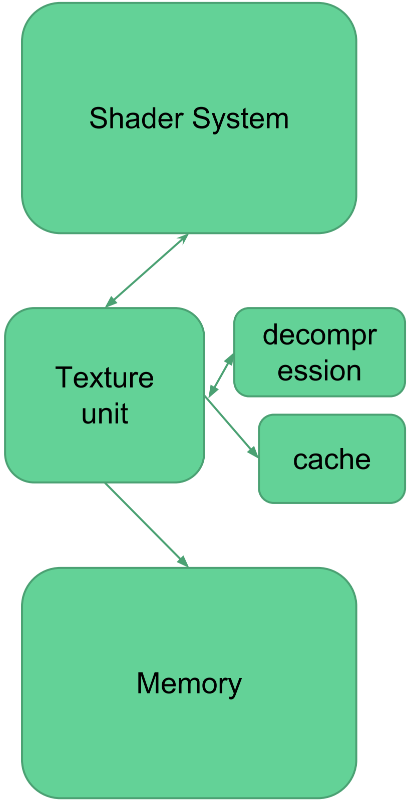
Pixel shading: a note about quads

- a quad may contain only one primitive.
  - In case the primitive does not touch 4 pixel , extra “ghost” are created
  - Ghost pixel are created alongside of the edge
  - Ghost pixel are necessary for derivative calculations
  - If the triangles is big enough this ia not a problem
    - We only have pixels across the edges
  - Bottleneck : Small triangles will create tons of this threads, lots of overshading.



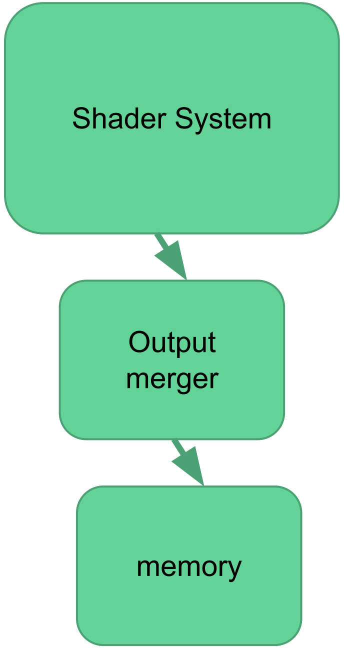
Texture unit

- Shaders usually need to access texture
- Request are performed by a texture unit
- A texture unit may serve more than 1 shader core.
- If the requested textel is not in the cache
  - It fetched from main memory
  - Usually very long latency… thousands of cycles
  - Imagine cooking something
  - But you need to walk to a shop a mile away each time you need an ingredient
- Perform texture interpolation
- Does Decompression
  - Some arch have compressed cache, some other don’t



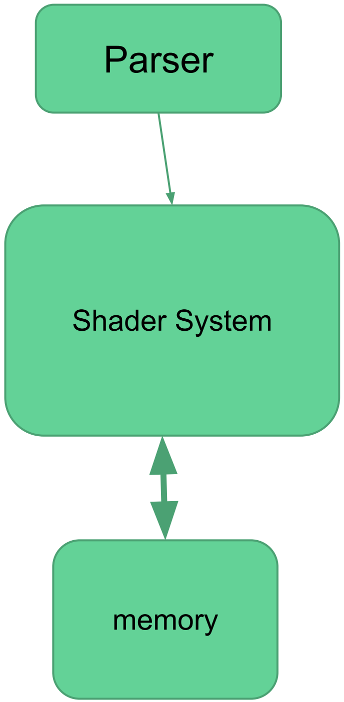
Output merger

- Also called Raster ops
- Export pixel color to render target(s)
- Write to main memory
- Also perform blend operations
- Limited number of pixel operation per clock
- Updates the Z buffer
    - “Late Z”
    - In Opengl/DirectX specs “Late Z” is the only stage for pixel rejection
    - Pixel needs to be exported in submission order (dx specs)
    - Also Late Z is the only Z rejection system that works if the pixel shader update z or using alpha mask



Compute shaders

- Used for Generic computation
  - Not bound to rasterization
  - So the command processor will send those command directly to the shader system
- Support to read/write textures and buffer plus atomics
- Shared local and global limited storage
- Can be asynchronous and run in parallel with graphics work

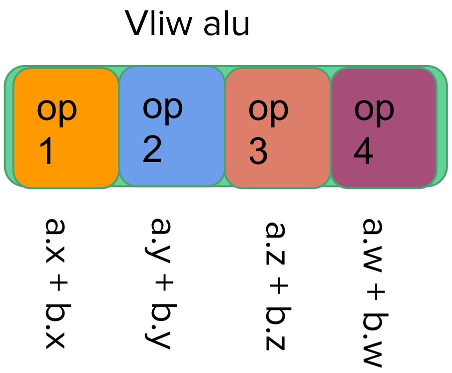


Shader Core

- The programmable part of the gpu
- There are many shadercore in a gpu
  - A lot of work can be done in parallel
  - Example: Geforce RTX 2080 has 2944 cuda cores
- Very simple unit compared to a CPU
  - In order execution
  - No speculation
  - No branch prediction
  - But very fast at context switching
  - Very good at latency hiding
- Multiple ALU shares program counter

VLIW architecture

- Very Long instruction word
  - Example: vliw4 (4 pipe vliw) means each core could do 4 independent instructions at the same time
- Maps very well with simple per pixel operation (dot, etc)
- Doesn’t map well with general programming
- Compiler need to statically schedule things to the vector pipes.
- Not always all the pipes can be used…not very efficient



Vliw example from Cayman architecture

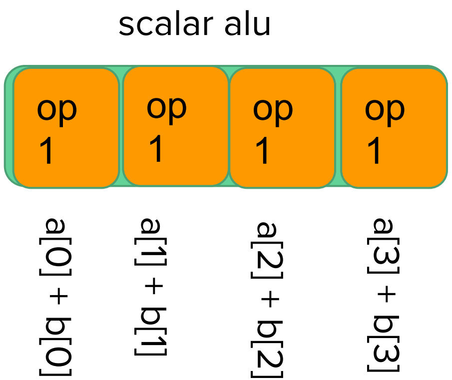
Table 4.1 Instruction Slots in an Instruction Group

Slot	Entry	Bits	Type
0	Scalar instruction for ALU.X unit	64	*src*.X and *dst*.X vector-element slot
1	Scalar instruction for ALU.Y unit	64	*src*.Y and *dst*.Y vector-element slot
2	Scalar instruction for ALU.Z unit	64	*src*.Z and *dst*.Z vector-element slot
3	Scalar instruction for ALU.W unit	64	*src*.W and *dst*.W vector-element slot
4	Scalar instruction for ALU.Trans unit	64	Transcendental slot
5	X, Y elements of literal constant (X is the first dword)	64	Constant slot
6	Z, W elements of literal constant (Z is the first dword)	64	Constant slot

Given the options described above, the size of an ALU instruction group can

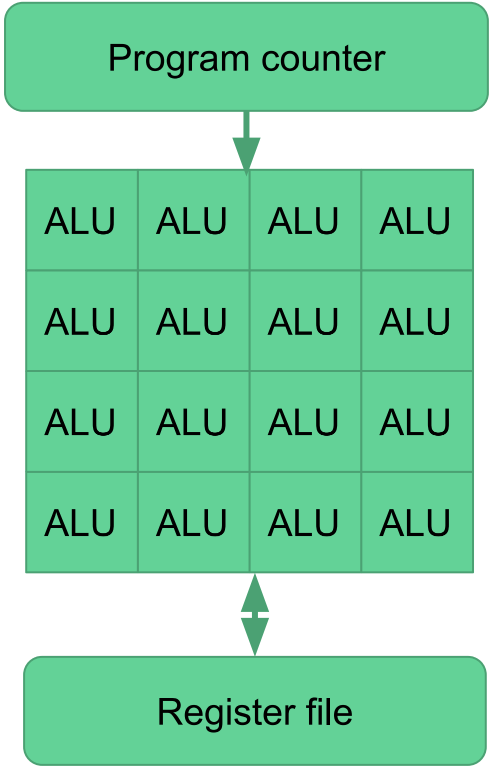
Moving to Scalar architecture

- Single instructions runs across a vector of data
  - It means you don't need to vectorize your code
    - The scheduler organizes vectors of data the instruction need to run
  - Example :Sum 8 float is equivalent to sum 2 float4
  - Concept similar to loop unrolling on CPU



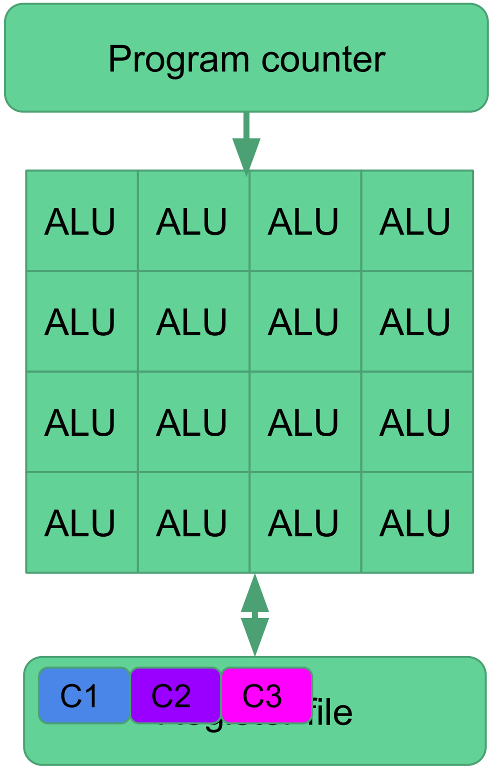
Fictional Shader Core

- 16 scalar alus , 16 instruction in parallel
- Each alu can do one 32 bit floating point instruction in one cycle
- one program counter
  - Same instruction is performed 16 times over 16 different data streams
- Register file is big enough for the 16 alus to work in parallel and perform context switching
- Each register have 16 slots, one for each thread



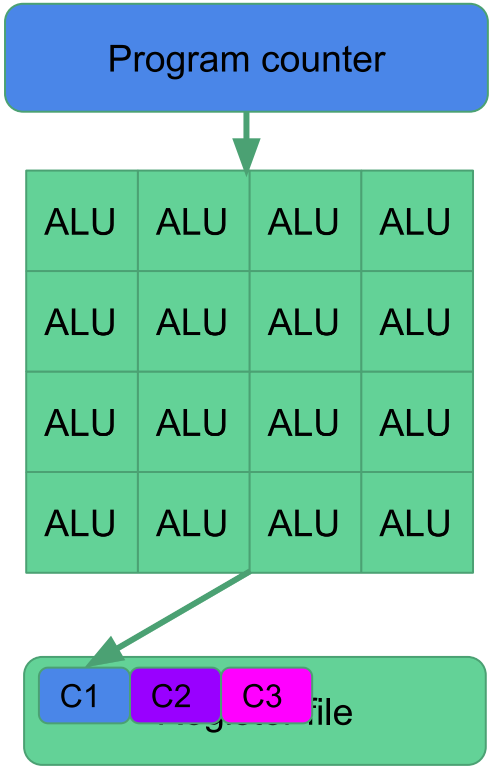
Fictional Shader Core -latency hiding-

- Cores need to access data in memory
- Accessing memory requires several hundreds of cycles
- During this period the alus have nothing to do
- However if the register file is big enough to contain multiple context, alus can switch to another thread
- If there are enough context , alu will not be idle.



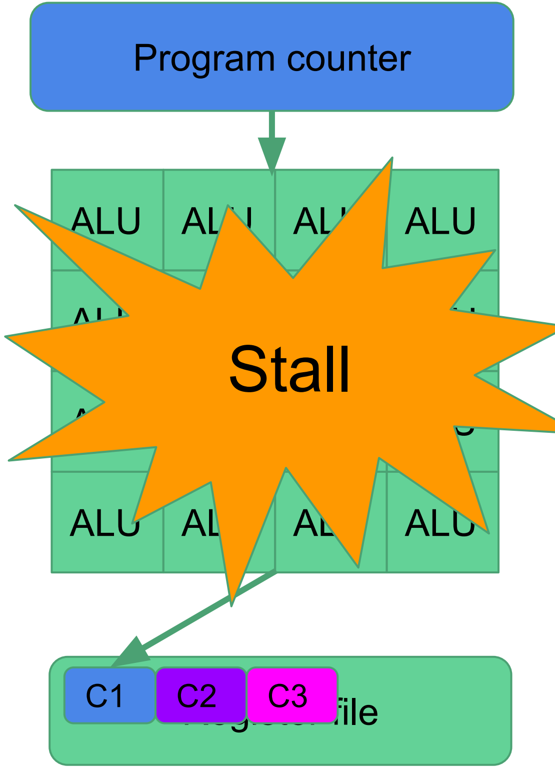
Fictional Shader Core -latency hiding-

- Example
- Alus are processing the C1 group of threads
- At some point there is a dependency stall



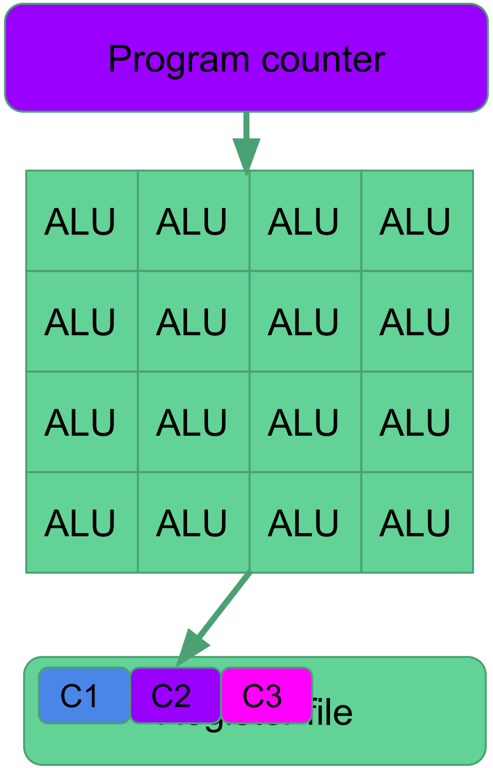
Fictional Shader Core -latency hiding-

- Example
- Alus are processing the C1 group of threads
- At some point there is a dependency stall
- It then switches to C2



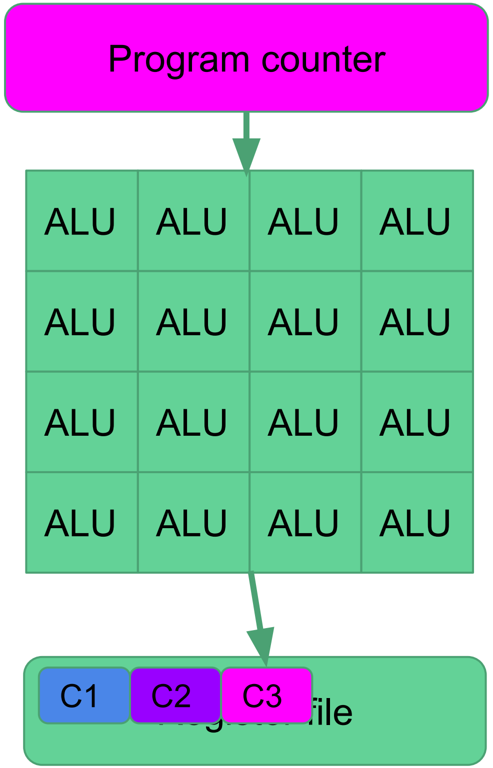
Fictional Shader Core -latency hiding-

- Example
- Alus are processing the C1 group of threads
- At some point there is a dependency stall
- It then switches to C2
- Eventually C2 will stall too



Fictional Shader Core -latency hiding-

- Example
- Alus are processing the C1 group of threads
- At some point there is a dependency stall
- It then switches to C2
- Eventually C2 will stall too
- Alus can switch to C3

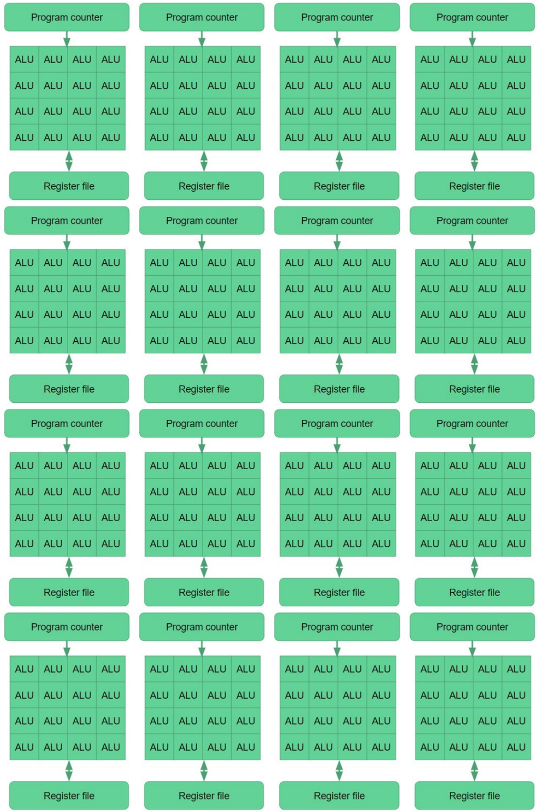


Fictional Shader Core - occupancy-

- The register file is dynamically partitioned
- “Big” shaders requires many registers
- And it will affect the number of concurrent context
- Space only for one context? No latency hiding :(
- Achieving maximum # of context is not fundamental
  - Usually memory bottleneck first
  - But need to be high enough to hide latency
- Example: A shader takes 100 registers, register file is 10kb
  - 10240 bytes /(16 alus * 4bytes (32bit)) = 160
  - 160 / 100 = 1.6 context :(

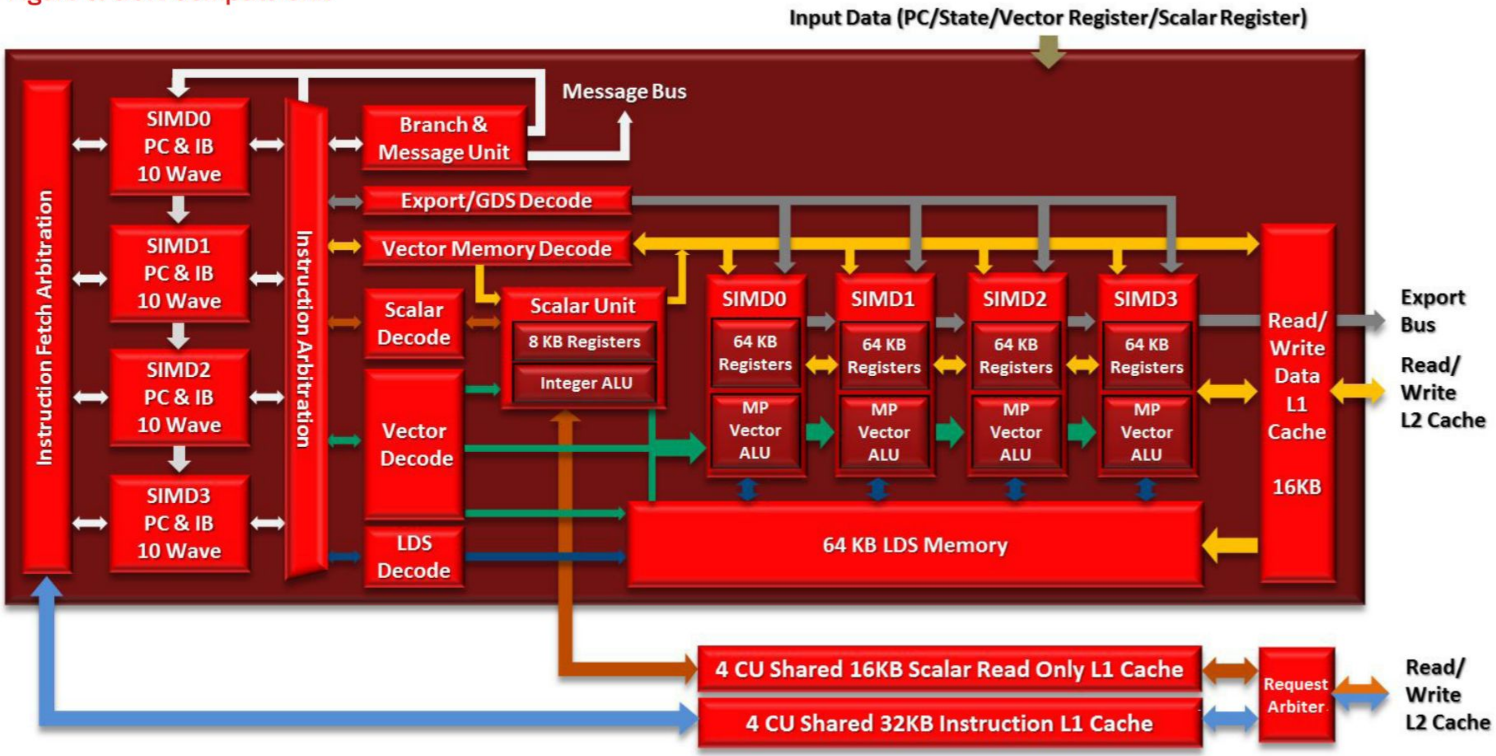
Fictional Gpu

- 16 cores
- Each core 16 alu
- 256 operations in parallel over 16 different instruction stream
- Clocked @1Ghz = 256 Gigaflop.
- Gpu stages are executed in parallel
- As soon as a triangle is transformed it is rasterized
- Each core can deal independently with pixel shaders, vertex shaders, compute and so on



Shader system use case : GCN

Figure 3: GCN Compute Unit



Wavefronts

- The smallest unit of work in gcn is a wavefront
- A wavefront is a group of 64 threads
- A thread is a single “instance” of the shader that work across only one data path/ lane
- Example:

```
void main()

    gl_FragColor = vec4(0.4, 0.4, 0.8, 1.0);
}
```

- A wavefront is 64 pixel worth of work
- A thread is 1 pixel inside a wavefront

VGPR / SGPR

- A vgpr is a register that has 64 32-bit entries
  - Imagine them uint_32 vgpr[64];
- An operation that takes a vpgr operands will happen on all the 64 entries simultaneously
- A SGPR instead is a register that is a single 32bit entries
  - Useful for operation that are constant across all the wavefronts, wavefront status flags and so on

example

```
void main()

    gl_FragColor = vec4(0.4, 0.4, 0.8, 1.0);
}
```

```
v_mov_b32       v0, 0x36663666
v_mov_b32       v1, 0x3c003a66
exp             mrt0, v0, v0, v1, v1 done compr vm
```

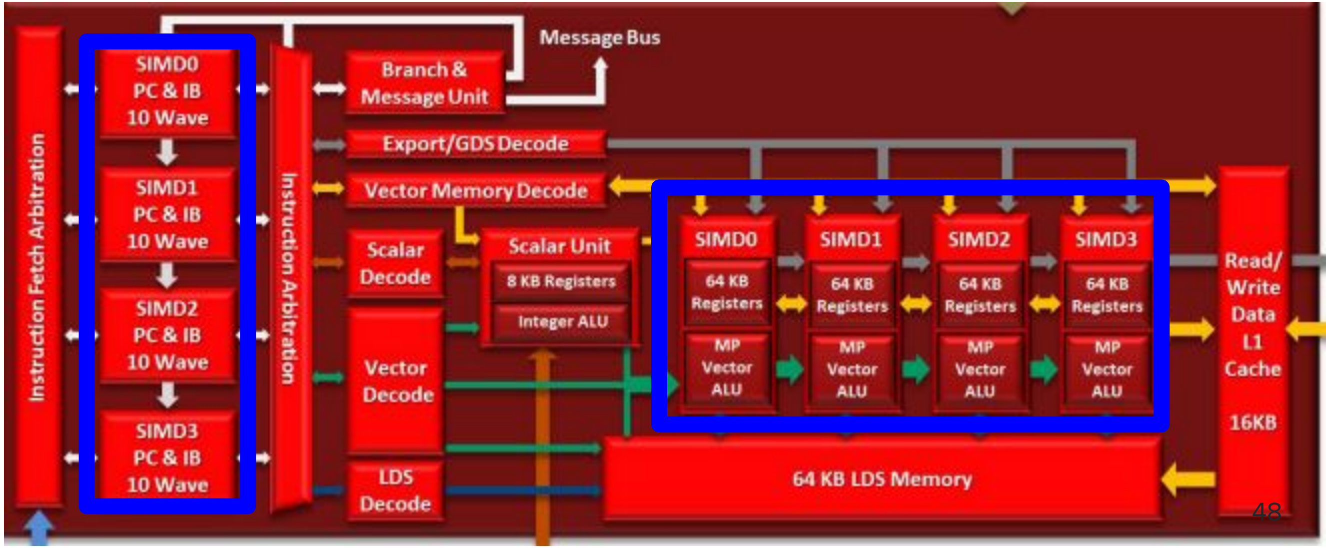
Move 0.4 into vgpr v0
Move 0.8 into vgpr v1
Export the pixel as v0 v0 v1
All of this happens 64 times simultaneously inside the CU

CU

- Smallest compitational unit
- A gpu contains many CUs
- A cu contains 4 simd unit
  - Each simd can execute an instruction on 16 different data (simd16)
- A scalar unit
- A branch unit
- 256kb for vector registers
  - 256kb /4 simd/ 64 lane = 256 vgpr
- 8kb for scalar registers
  - 256kb /4 simd = 512 sgpr

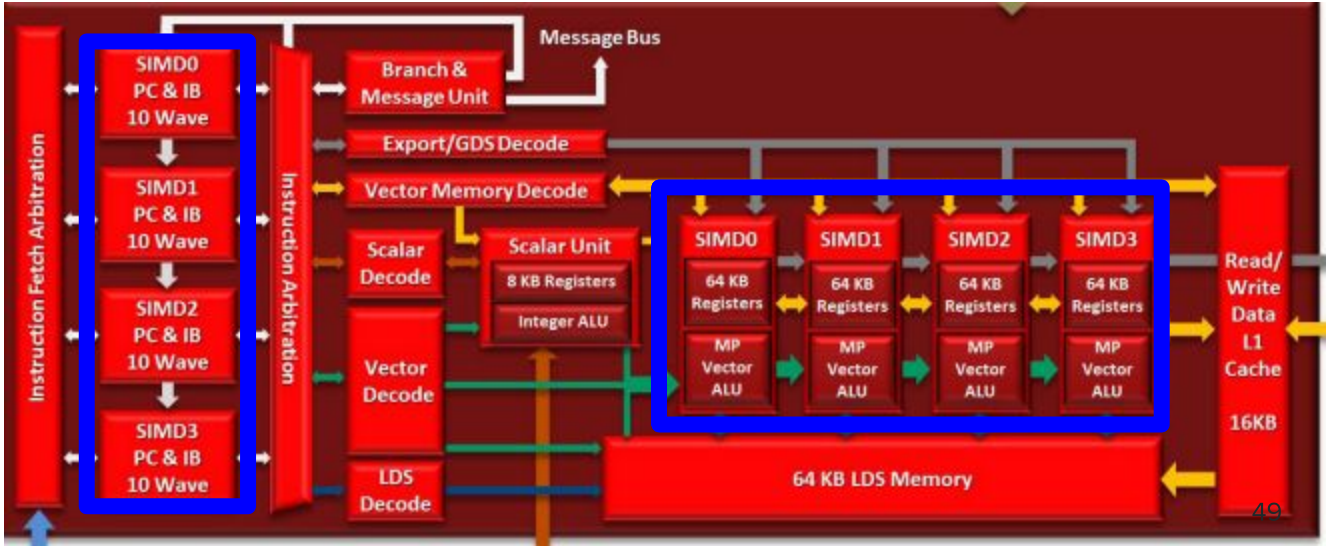
CU simd

- Each simd has its own program counter
  - Current instruction inside the wavefront
- Each simd can process 16 32bit values in 1 cycle
  - An entire wavefront takes 4 cycle to be processed by a simd



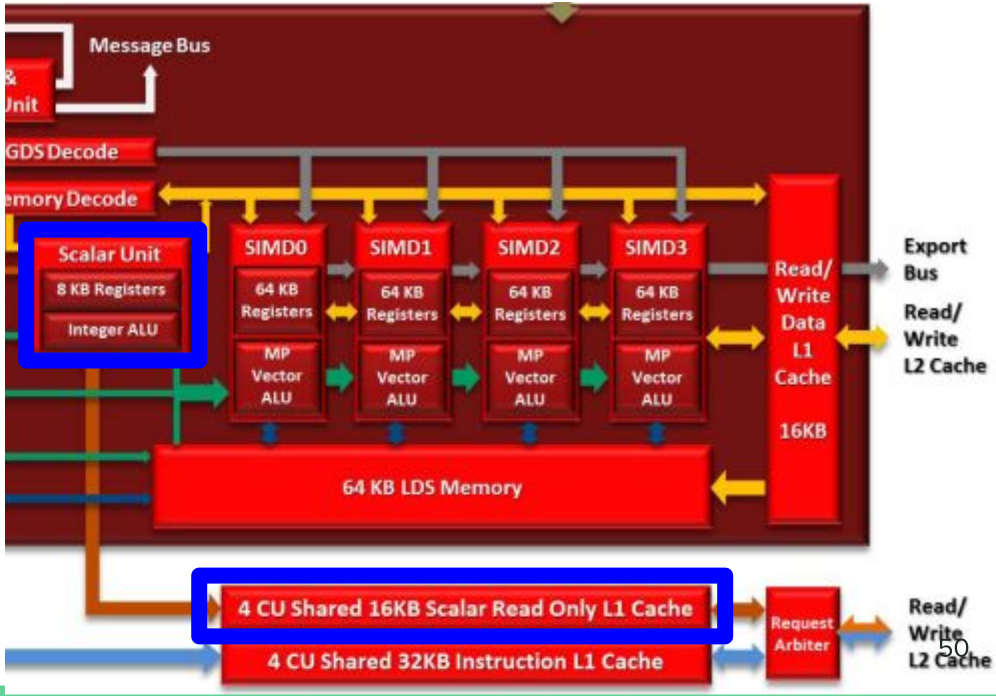
CU simd

- Each simd has an instruction buffer of 10 wavefronts
- Maximum 40 wave in flight per CU
- Depending on registry usage
- Potentially coming from different kernel/shaders



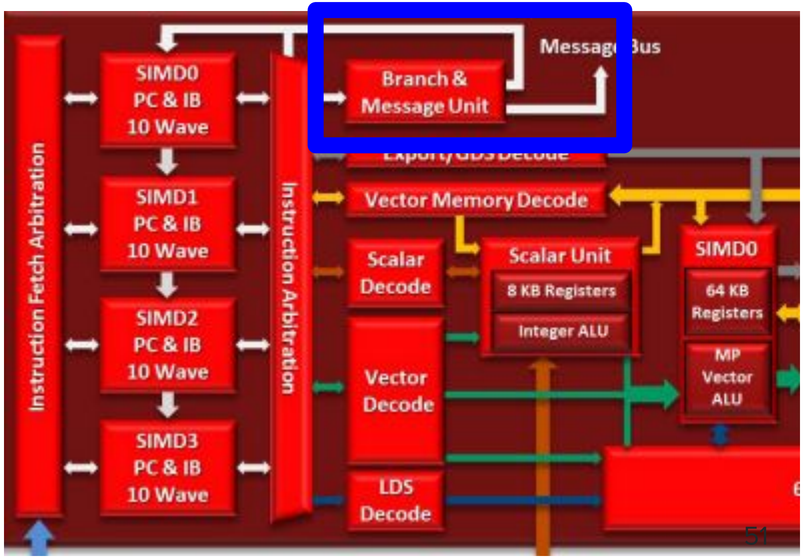
CU scalar unit

- Mainly for control flow across the wavefront
  - Ex if (constant_flag) then else
- Constants are taken from a read Only cache
- Also handling interrupts/synch
- Scalar operands operations



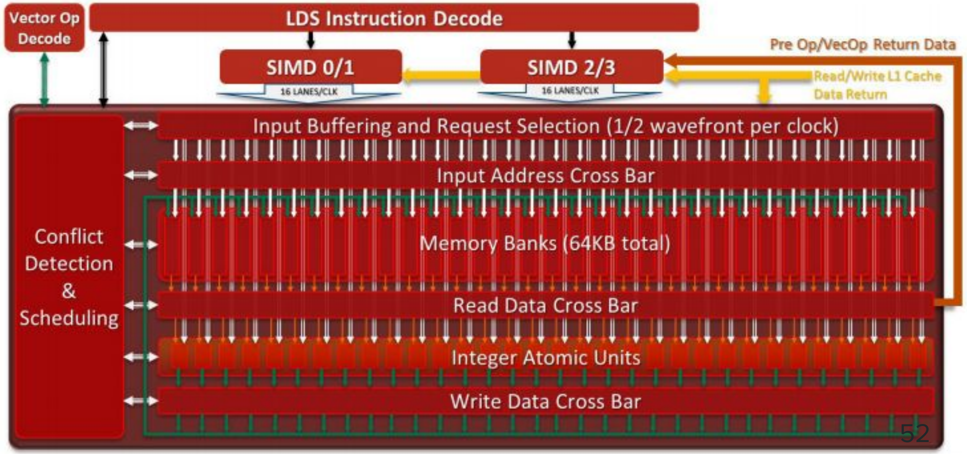
CU branch unit

- Handles vector branches
    - Ex : if vgpr > 0 then else
- Handles floating point exception
- Send message to other units/host cpu



LDS/GDS

- A CU have also a shared read/write memory of 64kb (LocalDataShare)
- Used by pixel shader as storage for interpolant
- But fully accessed by the programmer
  - Thread Group Memory
- Needs to handle atomic operation and thread group synchronization
- Example usage: caching texure data across a compute threadgroup
- GDS is shared across all the CU
- Can do ordered_cont op



Export

- When the program is finished it usually issue an export
- Always the case of a pixel shader
- It marks the end of the programmable part and pass down the data to fixed function block
    - Ex: export in a pixel shader pass the control over the color block

Vector memory

- CU have an internal 16kb L1 cache for vector memory operation
  - Usually texture data
- L2 is outside the CU

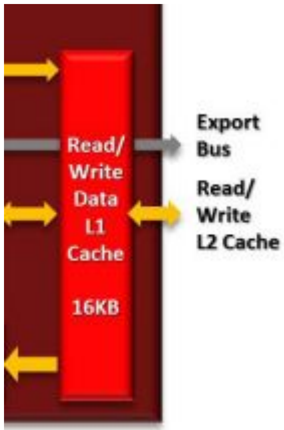
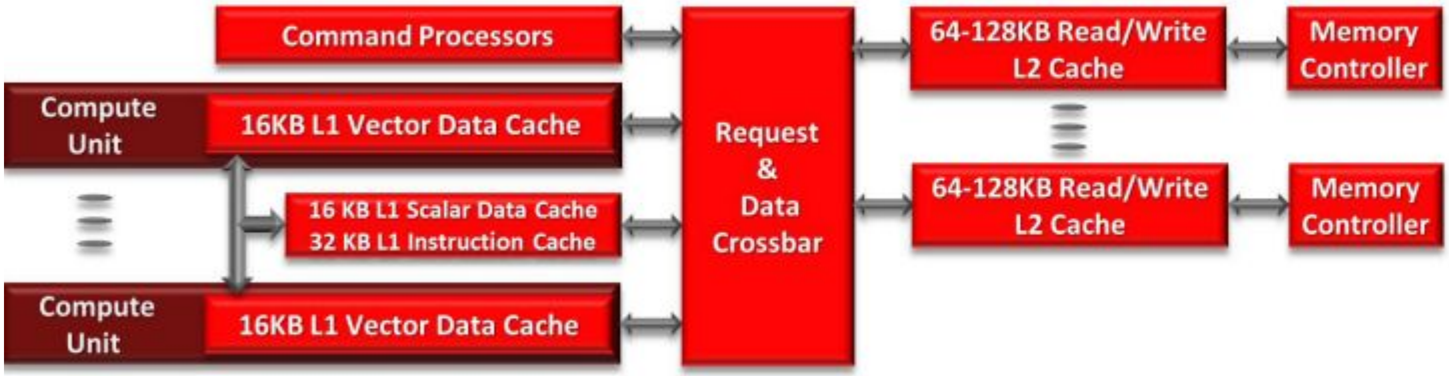


Figure 6: Cache Hierarchy

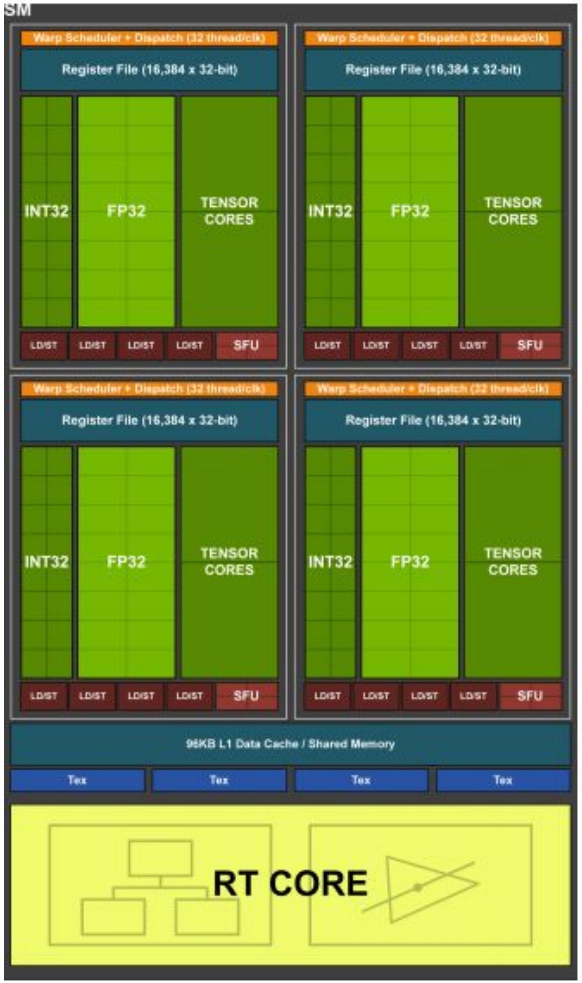


Shader system use case: Nvidia Turing(TU102)



Streaming multiprocessor (SM)

- An SM contains 4 group of 32 cores
- Each group have its own
  Instruction buffer,warp scheduler
  Register files
- 4 texture units /l1 cache and
  64kb shader memory
- Each core have 16 floating point unit
  16 integer unit and two tensor cores
- Each SM have a dedicated Ray Tracing unit
  For traversal / intersection

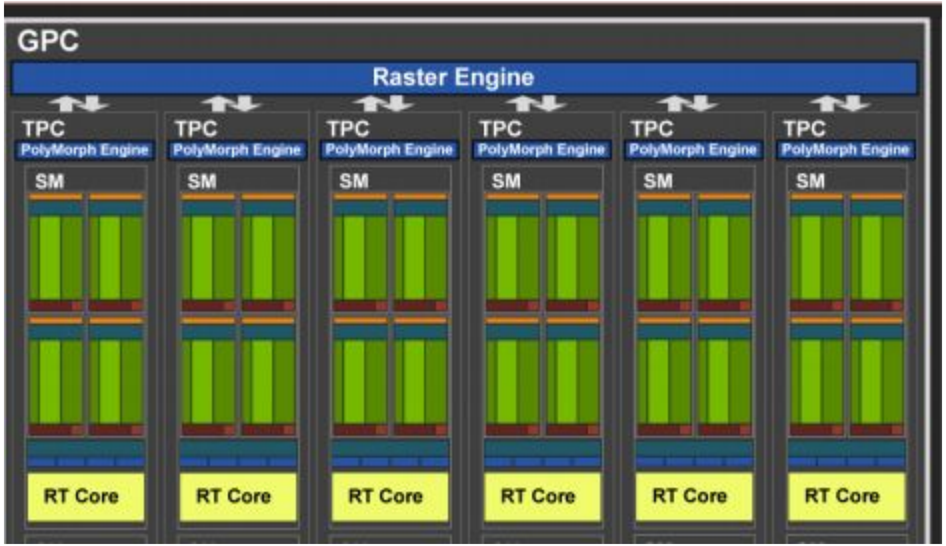


warps

- A warp is a group of 32 threads and is the smallest unit of work
- Each SM can hold 64 warps in flight
- Each thread can access a maximum of 255 registry
- Usage determine actual number of concurrent thread

Graphics processing cluster (GPC)

- Each GPC has:
- A Rasterizer
  - Turns triangles data into actual pixel
  - Ready to be dispatched as warp
  - Perform triangle and z culling
- 6 Texture Processor Cluster
  - 2 SM each
  - A polymorph engine
  - Perform vertex fetching and assembles Vertex warps
-

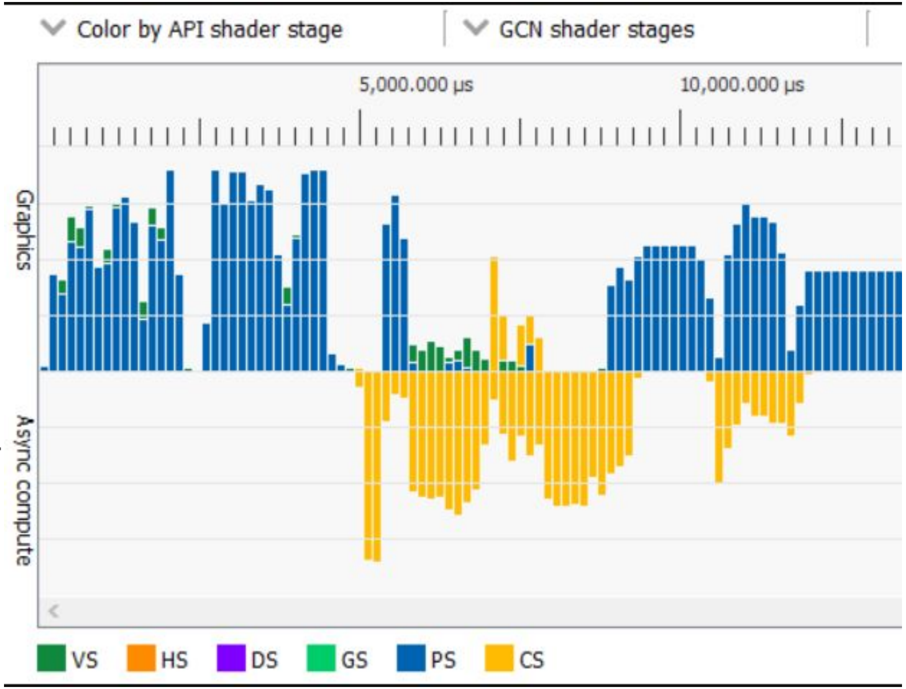


GeForce RTX 2080

GPU Features	GeForce GTX 1080	GeForce RTX 2080
Architecture	Pascal	Turing
GPCs	4	6
TPCs	20	23
SMs	20	46
CUDA Cores / SM	128	64
CUDA Cores / GPU	2560	2944
Tensor Cores / SM	NA	8
Tensor Cores / GPU	NA	368
RT Cores	NA	46

Summary

- A GPU is an extremely parallel machine
  - Each of the stages are executed as soon as there's enough work to do.
  - This is called immediate rendering
  - Shader system can have any kind of work in flight at a given time.
  - While the Rasterizers, PA, IA and output merger are processing other things.
  - Data dependency is a limiting factor.
- Knowing what happens in each stage can help spot the bottleneck.



Radeon gpu profile

About mobile GPUs

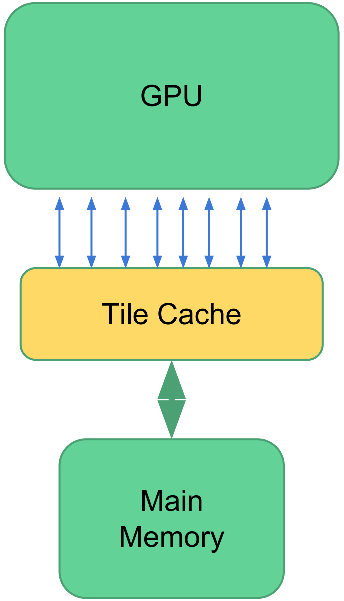


Problem on mobile

- Battery consumption is king
- Very high bandwidth memory system is power demanding
- Low bandwidth is “slow”
  - 10x slower than mobile
- Solution : Tiled based / Tile based deferred architectures
  - TBR/TBDR for short

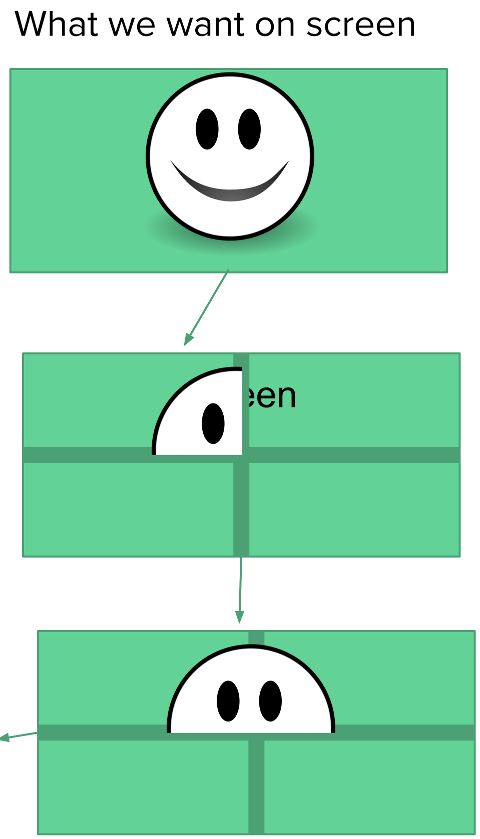
Tile Based architecture (I)

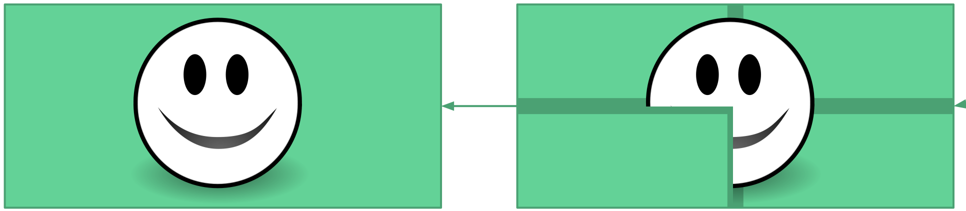
- Use of a hi speed on chip cache.
- Used as temporary storage during vertex/pixel shading.
- Main Memory can be low bandwidth.
- Only “invoked” when writing the final pixel data.
- An for texture access.



Tile Based architecture (II)

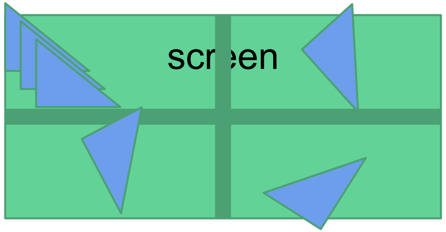
- To maximize cache usage screen is divided in tiles
- Screen is renderender one tile at the time
- Tile cache is used at temporary store for the framebuffer
- When finished the content of the tile is written back to memory





Tile Based architecture (III)

- We need to sort all the triangles in tiles
- We need to pre process all the geometry first
- All vertex shaders runs first
- Then we know triangles per tile
  - “Binning”
- Binning happens in main memory
  - Based on principle that “normally” there are less triangles than pixel
- On a desktop GPU pixel and vertex runs in parallel



Tile Based Deferred architecture

- Once all triangles are sorted in tiles pixel processing can start
- Since we know all the primitive on tile we can pick only the one that contribute to pixel color
  - Example: nearest one
- If this happen the architecture is said to be “deferred”
- Great reduction of pixel shading work
- Only runs shaders that actually write a pixel
- On a desktop GPU this is possible with “Z prepass”
  - But need to submit the geometry twice

TBR pro and cons

- Pro
  - Frame buffer bandwidth reduces
  - Z prepass “free”
  - Tiled cache more efficient than cache lines
  - Blending happens in the tile cache
    - Programmable blending possibile
- Cons
  - Split rendering in two, lockstepped, stages
  - Tile cache limits usage of frame buffer format and multiple render target
  - Complex scene might slow down heavily the binning process
  - Harder to read cross tile pixels

Gpus- where are we now

- Mobile and desktop fundamentally different.
- Proprietary features to optimize the vertex pipeline
  - Nvidia - mesh shaders
  - Amd - primitive shaders
  - VR rendering vertex optimization
- Proprietary features to optimize rasterization
  - Native variable shading for foveated rendering
  - Tile rendering similar to Mobile architecture
  - Mentioned in vega white paper
  - Nvidia experiment https://github.com/nlguillemot/trianglebin
- Raytracing
  - Really hard problem to solve
  - Dynamic bhv creation
  - Gpu traversal
  - Handling incoherent rays

Conclusions

- Basics of both Desktop and mobile architecture covered
- Highlighted possible bottlenecks for each stage
- We can use this knowledge to understand profiling
- GPU evolved over the years
    - Always know your target architecture!

References

Realtime rendering 4th edition

scratchapixel.com

A trip through the Graphics Pipeline 2011

AMD GCN whitepaper

AMD Vega whitepaper

Nvidia turing whitepaper

PoweVR Recommendation

Samsung Developer Website